Applying Formal Methods to an Information Security Device: A Case Study*

Presented at the NATO Symposium on Protecting Information Systems in the 21st Century, Washington, DC, October 25-27, 1999

James Kirby, Jr. **Myla Archer**
Constance Heitmeyer
Code 5546, Naval Research Laboratory, Washington, DC 20375
{kirby, archer, heitmeyer}@itd.nrl.navy.mil

Abstract

One approach to assuring information security is to control access to information through an appropriately designed device. A cost-effective way to provide assurance that the device meets its security requirements is to detect and correct violations of these requirements at an early stage of development: when the operational requirements are specified. Once it is demonstrated that an operational requirements specification is complete and consistent, that it captures the intended device behavior, and that the operational specification satisfies the security requirements, this operational specification can be used both to guide development of implementations and to generate test sets for testing implementations. This paper describes the application of the SCR (Software Cost Reduction) requirements method and the NRL's SCR toolset, which includes a set of verification and validation tools, to a US Navy communications security device. It reports on our success in proving that the operational requirements specification satisfies a set of security properties. The paper also discusses the practicality and cost of applying formal methods to the development of security devices.*

1 Introduction

Many studies (e.g., [5]) have emphasized the importance of discovering and eliminating flaws in a system at the earliest possible stage of development: the requirements phase. The application of mechanized formal methods can expose many errors that humans miss in inspecting even the most carefully crafted requirements specifications [22]. A *formal method* is a mathematically-based method for the precise specification and analysis of systems and devices. Associated with a formal method is a formal specification language with a well-defined semantics. A *mechanized* formal method is one having, at the very least, computer support for checking specifications for well-formedness, i.e., conformance to the syntactic and semantic restrictions of its specification language. To provide increased confidence in the correctness of the requirements specification, formal techniques which automatically check the specification for critical application properties, such as security properties, are also valuable. Such techniques may be encoded directly as part of the mechanized formal method, or they may be provided by an interface to an existing technique, such as a mechanical theorem prover.

Prior to analysis by mechanized formal techniques, a requirements specification must be represented in some formal specification language. Sometimes, this representation combines an operational requirements specification—e.g., a specification that represents a system as a state machine—with a property-based specification describing the required system properties. Sufficiently detailed requirements specifications, even those written in prose, often lend themselves to representation in this form. Once this representation is obtained, one can apply mechanized formal techniques to ensure that the operational requirements specification is self-consistent and complete, that it represents the customer's intentions, and that it satisfies required system properties.

Assurance that a system specification satisfies a critical set of properties is especially important for applications that must perform mission-critical operations safely and securely. Equally important is assurance that the system implementation conforms to the specification. One approach to obtaining such assurance is incremental refinement of the operational specification into source code (or its hardware analog, a silicon chip). Another approach uses the operational requirements specification as the basis for generating sets of test scenarios for an implementation. To provide confidence that the scenarios test examples of all "possible" system behaviors, these scenarios must satisfy some coverage criterion.

The SCR (Software Cost Reduction) requirements method [14, 11] is a tabular formal method for specifying the black-box behavior of a system, i.e., its operational requirements. The use of SCR is supported by the SCR* toolset [13, 10, 11] developed at NRL. SCR* is designed to support engineers in the development of real systems. Besides providing an editor for creating SCR specifications, SCR* supports the analysis of SCR specifications with a wide range of techniques, including consistency checking, simulation, invariant generation, model checking, and theorem prov-

*This work is funded by the Office of Naval Research and SPAWAR.

ing. Analysis using SCR* is fast, easy, and, to the extent possible, "push-button" (i.e., automatic) [11]. Engineers using SCR*'s push-button analyses do not need specialized knowledge to use the analysis techniques. SCR* also provides easy-to-understand feedback. Additional tools planned for the SCR* toolset are a test set generator (now in the prototype stage) and a tool to generate source code from specifications.

This paper reports on the application of SCR to a subset of the requirements specification of a real Navy COMSEC (communications security) device, CD, based on the Programmable Embeddable INFOSEC Product (PEIP) technology. The paper describes how SCR was used in a cost-effective manner to create a high-quality SCR specification for CD and to prove that this specification satisfies a set of security properties and fails to satisfy one property. It illustrates how the SCR* analysis techniques both complement and support one another, and how one particular technique, the TAME [1, 2] interface to the PVS [26] theorem prover can be used both to prove security properties and discover property violations.

After reviewing the SCR method and the SCR* toolset, Section 2 introduces PEIP and the COMSEC Device CD. Section 3 describes the translation of a prose requirements document for CD into an operational SCR specification and the application of tools in the SCR* toolset to the analysis of this specification. Section 4 discusses our results, the utility of the individual tools, and the interactions among the tools. Finally, Section 5 discusses related work, and Section 6 presents our conclusions and future plans.

2 Background

2.1 The SCR Method

The SCR method (Software Cost Reduction) is a formal method for specifying and analyzing the requirements of safety-critical control systems. Since its introduction in 1978, the SCR requirements method has been applied successfully to a wide range of critical systems (see, e.g., [14, 23, 7, 6, 22, 18]), including avionics systems, space systems, telephone networks, and control systems for nuclear power plants.

An SCR requirements specification describes both the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [13, 12]. In the SCR model, the system environment and required system behavior are respectively described by NAT and REQ, two relations of the Four Variable Model [24]. NAT describes the natural constraints on the system, such as constraints imposed by physical laws and the system environment. The system environment contains *monitored quantities* that the system monitors and *controlled quantities* that the system controls. REQ describes the required relation between the monitored and controlled quantities that the system must enforce. The monitored and controlled quantities are represented in the SCR model as *monitored and controlled variables*. To specify REQ concisely, SCR specifications also use two types of auxiliary variables, *mode classes*, whose values are *modes*, and *terms*, both of which often capture historical information.

In the SCR model, the environment nondeterministically produces a sequence of input events, where an *input event* signals a change in some monitored quantity. The system described in an SCR specification is represented as an automaton—i.e., state machine—that begins execution in some initial state and responds to each input event in turn by changing state and by producing zero or more output events, where an *output event* is a change in a controlled quantity. In this representation, the system behavior is assumed to be *synchronous*: the system completely processes one input event before the next input event is processed. The system states are determined by the values assigned to the system variables, i.e., the monitored variables, controlled variables, mode classes, and terms. An SCR specification defines the set of initial states and the transition relation of the automaton.

An important concept underlying the notion of a system property to be satisfied by a specification is *reachability*. A *reachable state* of the automaton either is an initial state or can be reached from an initial state by a finite sequence of transitions. A *reachable transition* of the automaton is a transition from a reachable state. A *state invariant* of the automaton is a predicate on the state variables that evaluates to **true** in every reachable state; a *transition invariant* is a predicate on the variables of two states that evaluates to **true** for the two states in every reachable transition. Our experience with practical systems is that most critical system properties can be represented as either state invariants or transition invariants.

The transition relation of an SCR automaton is defined in terms of conditions and events, where a *condition* is a predicate defined on a system state, and an *event* is a predicate defined on two system states implying that they differ in the value of at least one state variable. When the value of a variable changes, we say that an event "occurs". The notation "`@T(c)`" denotes an event, and is defined as $\texttt{@T(c)} \stackrel{\text{def}}{=} \neg \texttt{c} \wedge \texttt{c}'$, where the unprimed condition `c` is evaluated in the *current* state and the primed condition $\texttt{c}'$ is evaluated in the *next* state. Informally, "`@T(c)`" means that condition c becomes true and "`@F(c)`" means that `c` becomes false. The notation "`@T(c) when d`", where `c` and `d` are conditions, denotes a *conditioned event*, and is defined as $\texttt{@T(c) when d} \stackrel{\text{def}}{=} \neg \texttt{c} \wedge \texttt{d} \wedge \texttt{c}'$.

An SCR specification contains a set of tables which describe the state transitions, that is, how to compute the values of individual variables in the next state.

Each controlled variable, term, or mode class has a corresponding table. The table for a mode class is a *mode transition table*, which maps a source mode and an event to a destination mode. The table for any term or controlled variable is either an *event table*, which maps conditioned events to values of the variable in the next state, or a *condition table*, which maps conditions on the next state to values of the variable in the next state. Examples of an event table and a condition table as they appears in the SCR* specification editor are shown in Figures 1 and 2. For an example of a (partial) mode transition table, see Figure 8 below.

Figure 1. Event table for `cKeybank1Key1`

Figure 2. Condition table for the controlled variable cAlarmIndicator of CD.

Besides the tables for each variable, an SCR specification contains dictionaries of *types*, *variable declarations*, *constant declarations*, *environmental assumptions*, and *specification assertions*. The first four dictionaries describe more about the the SCR automaton being specified. The specification assertion dictionary records the properties that the system designer wishes the system to have. For CD, these properties include the security requirements. Figure 3 shows part of the specification assertions dictionary for CD. An important function of SCR* is to determine whether the properties in the specification assertions dictionary are satisfied by the operational specification defined by the other dictionaries and tables.

Figure 3. A portion of the specification assertions dictionary for CD.

2.2 The SCR* Toolset

The SCR* toolset [13, 10, 11] is a set of software tools developed by NRL to provide mechanized support for the SCR method. In addition to a *specification editor* for creating and modifying a requirements specification and a *dependency graph browser* to display the dependencies among the variables in the specification, the toolset includes an automated *consistency checker* to detect type errors, missing cases, circular definitions, and other types of application-independent errors; a *simulator* to allow users to symbolically execute the specification to ensure that it captures their intent; an interface to the *model checker* Spin [15] to detect property violations; and an *invariant generator* that computes state invariants from the tables describing the state transitions. To provide formal underpinnings for the tools and for the analysis techniques the tools implement, a formal model has been developed to define the semantics of SCR requirements specifications [12, 13].

Several additional tools have been integrated with SCR* by means of translators that transform the internal representation of an SCR specification into the input languages of the tools. These tools include *TAME* (Timed Automata Modeling Environment) [1, 2], an interface to the theorem prover PVS [26] that simplifies using PVS to prove properties of automata models, a *validity checker* [4] that automatically checks whether a predicate over the variables of one or two states is a state or transition invariant of an SCR specification, and a *test set generator* [8] that automatically generates test sets from an SCR specification. Section 3 summarizes the results of applying SCR* tools for verification and validation to the SCR representation of CD.

2.3 The PEIP COMSEC Device

PEIP (Programmable, Embeddable INFOSEC Product) is a technology for building communication security (COMSEC) devices. Unlike most other communication security devices, PEIP devices will contain software as well as hardware. While the security community has significant experience in providing a high level of assurance that hardware COMSEC devices behave correctly, experience providing such assurance for COMSEC devices containing software is rare.

CD (COMSEC Device) is a communications security device, based on PEIP technology, which provides cryptographic processing for a US Navy radio receiver. CD will generate keystreams compatible with another cryptographic device and will support multiple channels. Because it is programmable, CD will allow the Navy to use more than one cryptographic algorithm. Among other capabilities, CD can download associated algorithms and keys into working storage, assign them to designated communication channels, maintain the association between an algorithm and its keys, clear algorithms and keys from memory when required, and generate keystreams.

The CD System Requirements document, referred to below as the CD SRD, is a detailed prose document describing the requirements for CD, including the required operating states and modes and the functional, design, and implementation (e.g., materials and workmanship) requirements. The requirements specification in the CD SRD was designed to satisfy a large set of security requirements that are required for certification by the DoD organization that evaluates INFOSEC devices and certifies them for use.

For an informal prose document, we found the CD SRD to be relatively complete, consistent, and precise, with the exception of the descriptions of the functions for generating keystreams and for cryptographic synchronization which were intentionally left incomplete. Our method detected minor inconsistencies in the description of required states and modes in the requirements document.

3 Applying SCR* to CD

This section discusses the translation of a subset of the prose specification provided by the CD developers into an SCR specification and the application of the tools described above to the SCR specification. In particular, we present the results of applying the SCR* consistency checker, SCR* simulator, and several tools that generate or verify properties of SCR specifications and describe the future use of the SCR* testing tool to generate test sets from the SCR CD specification. All examples showing results of SCR* analyses reflect properties of our SCR specification of CD.

3.1 From Prose to SCR Requirements

The CD SRD is a traditional 2167A-style document. The section entitled "System Requirements" describes the requirements imposed on the behavior of the system. The subsection "Operating States and Modes" describes the system modes and transitions between them, and the subsection "CD Functional Requirements" gives further details on mode transitions and the functions that CD is required to perform. CD Functional Requirements are organized by function, where a *function* is a task that the system performs (e.g., key load function, reset function, report status function).

To develop the SCR specification, we studied the CD SRD, focusing on the constraints it imposed on the required system behavior, and representing those constraints using SCR constructs. The CD SRD was sufficiently precise and complete about key and algorithm management, modes of operation, and security requirements relating to power, tampering, and zeroizing for us to capture the required behavior in the SCR CD specification. We obtained security properties by examining the SCR specification and surmising the goals of the required behavior and by interpreting descriptions of functions in the CD SRD as security requirements (as well as behavioral requirements). The CD SRD intentionally did not contain sufficient precise information about cryptographic synchronization and generating keystreams for us to capture their behavior in the SCR CD specification. The CD project manager has reviewed our set of security properties and confirmed that, except for one, they are reasonable security properties of CD.

Most of the effort spent in building the SCR CD specification took place over a nine-month period as a background activity. The initial build of the specification took approximately one person-week. About one additional person-week was devoted to refining and completing the specification. Aside from our use of the consistency checker to find and correct errors, these efforts are distinct from those reported in this paper applying SCR* simulation and formal analysis techniques to the CD specification.

Much of the CD SRD is consistent with the SCR model of black-box requirements. Outputs (such as indicator lights and status messages) and inputs (such as the status of primary and backup power, data provided by the host, and positions of switches) fit the SCR model well. CD's operating states and modes are easily represented as an SCR mode machine.

However, there is much important CD behavior that does not fit easily into this model. The CD SRD describes at great length the rules for loading algorithms and keys, associating them with channels, and clearing them from memory. Such rules, which concern managing data in CD's memory, do not fit SCR's model of black box requirements: Memory is internal to the black box, whereas outside the black box, it is invisible. Since there is not enough information in the CD SRD to specify the rules for generating the key stream,

which is the reason the Navy uses CD, much required behavior that would be relevant and useful to reason about cannot be captured with a straightforward application of the SCR model. We resolve this dilemma by representing the manner in which CD is required to manage algorithms and keys in its memory as externally visible. The SCR CD specification defines controlled variables that represent the memory locations in which the CD software can store algorithms and keys.

There are an unlimited number of algorithms and keys that can be distributed among a number of algorithm and key storage locations and a number of channels. Since SCR does not yet have a capability for conveniently and concisely specifying a number of identical instances, the SCR specification assumes that there are at most two of any physical quantity that can have more than one instance. Each of the two instances is specified separately. Thus, for example, the SCR CD specification includes a variable declaration and a value function for each of two key banks and for each of the two key storage locations in each key bank. The specification also assumes there can be at most 1,000 different algorithms and 1,000 different keys.

Another feature of the CD specification that does not fit the SCR model is the Built-in Test (BIT). SCR views the Built-In Tests described in the CD SRD (and in many other requirements documents) as a design for determining the health of CD's security components, rather than as a requirement. The SCR CD specification replaces the full and background BITs by the monitored variables mHealthyFull and mHealthyBackground, which each denote the operational "health" of CD's security-critical components. The BITs described in the CD SRD are viewed as a design that should satisfy these requirements.

The SCR CD specification has one more mode than the original. SCR adds an Off mode so that the system is always in exactly one mode.

Our SCR specification, which models a subset of CD's complete operational requirements, contains 39 variables, specifically, 17 monitored variables, one mode class, two terms, and 19 controlled variables. Figure 4 shows the variable dependency graph for the complete SCR specification of CD. Variables are represented as boxes, and an arrow from one variable to a second variable indicates that the first variable depends on the second—or, more precisely, that the value of the first variable in the next state depends on the value of the second variable in either the current state or the next state. The heavy lines are backarrows; the number of backarrows reflects the complexity of the interdependencies among the variables, which is also reflected in the complexities of the tables. Although this graph has cycles, the SCR* consistency checker was used to assure that there were no circular dependencies among the "next-state" variables (see Section 3.2).

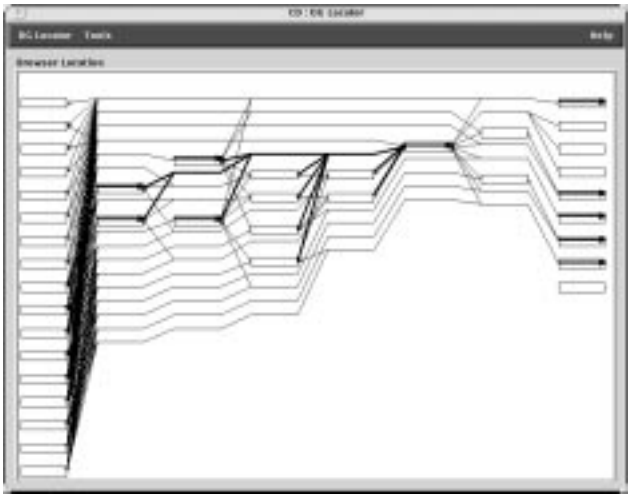

Figure 4. Full dependency graph for SCR CD.

3.2 Applying the Consistency Checker

The consistency checker analyzes a specification for consistency with the SCR requirements model. A form of static analysis, consistency checking avoids the computationally expensive activities of executing the specification and reachability analysis. The checks expose syntax and type errors, variable name discrepancies, unwanted non-determinism (called *disjointness errors*) missing cases (called *coverage errors*), and circular definitions (i.e., cycles in the dependencies among next-state variables). We designed the consistency checker to support engineers focused on developing high assurance systems. The checks are fully automatic push-button analyses that require no user input or guidance. When an error is detected, the consistency checker facilitates error correction by providing detailed feedback in the language of SCR. For some types of errors (e.g., non-determinism, missing cases), the checker will, in addition to describing the error and highlighting where in the specification the error occurs, provide a counterexample, a description of a situation that manifests the error.

In developing the CD specification, we frequently used the consistency checker for "sanity" checks. All of the checks but those for missing cases and non-determinism execute quickly enough that it is useful to invoke them many times during an editing session. We used the more computationally expensive checks for missing cases and non-determinism less frequently, because checking the entire CD specification for these properties requires 5 to 9 minutes.

The table in Figure 5 provides a short sample of errors reported by the consistency checker from the beginning of work on formalizing the CD specification in SCR. Each row of the table lists the error reported by the tool, what part of the specification is highlighted at user request, and our interpretation of the error. The cycle among the next state dependencies occurred while we were first trying to model in SCR certain com-

Error Report	SCR* Highlights	Diagnosis
Specification Assertion Dictionary (dictionary item aZeroKeysonTrouble): Type ERROR: Assertion expression is not a legal boolean in the Expression field	expression	In expression, an integer is illegally used as a boolean
smOperation Mode Transition Table: Cycle Detection ERROR: Cycle #1: Table smOperation uses mode class smOperation in the Name field; Function is smOperation Mode Transition Table	name of mode class in mode transition table	Events in transitions involving a mode class introduce a cycle in the next state variable dependencies
smOperation Mode Transition Table: Disjointness ERROR: Overlap between row 8 column 1 and row 9 column 1	two mode transition events that are not disjoint	A disjointness error is detected due to the illegal cycle and a missing variable definition

Figure 5. Consistency checker feedback.

plex mode transitions described in the CD SRD.

3.3 Simulating the CD Specification

Verification demonstrates that a specification has certain useful properties (e.g., deterministic, safe, secure). Although SCR* provides several verification tools (which we will describe later), the SCR* simulator is, instead, a tool for *validation*. Validation, which is especially important to users, demonstrates that the system, if implemented to conform to the specification, can be used in the intended environment and will be useful. It is typically difficult to validate a system before it is built, because the people who have the application knowledge required to judge the suitability of the system in the intended environment often do not have the skills required to read and analyze precise specifications. The SCR* simulator allows users to evaluate the behavior of the specified system before it is built and without reading the SCR specification.

Users can validate a system specified by an SCR specification by running usage scenarios through the SCR* simulator. The simulator's standard generic interface presents the state of the execution in terms of the values of the variables in the specification. The simulator separately lists the values of the monitored variables, controlled variables, terms, and mode classes. Each element of the usage scenario, a list of input events, assigns a new value to one of the monitored variables. When the simulator accepts an input event, it updates the values of the dependent variables (i.e., controlled variables, terms, and mode classes) before accepting the next input event. In addition to presenting the current state of the execution, the simulator can present a history of the execution.

Because this generic interface is most suitable for use by computer scientists and engineers, the simulator also supports the rapid construction of front-ends customized for particular applications. Instead of directly interacting with the values of variables in the specification, the user can interact with graphical representations of the entities in the application domain. E.g., instead of the simulator displaying the expression

```
cAlarmIndicator = on
```

to indicate to the user that an indicator labeled ALARM is lit, the simulator displays a graphical representation of a red light labeled ALARM. By interacting with such front-ends, the user moves out of the world of requirements specification and into the world of his application. Users who are expert in an application can use the application-specific front-end to validate the behavior captured by the specification before the system is built. They need not read the detailed SCR specification nor use the abstract generic simulator interface. Additionally, the simulator reports when a scenario violates specified properties.

We found an application-specific front-end for CD useful in interacting with the CD project manager. After seeing a simulation of CD using the CD-specific front-end (built in less than a day), the CD project manager provided us with useful feedback on the SCR specification of the CD. Hence, evaluation of the CD specification through this front-end to the simulator allowed a very effective use of a very scarce commodity, the project manager's time.

3.4 Automatic Invariant Generation

The SCR* invariant generator is based on an algorithm for generating state invariants from the functions defining the dependent variables, i.e., variables other than monitored variables. The algorithm computes invariants for variables defined by mode transition tables, event tables, and condition tables, and thus covers all of the dependent variables. For a dependent variable v taking values in a finite set $\{a_1, a_2, \ldots, a_n\}$, the algorithm generates for each a_i an invariant of the form

$$(v = a_i) \Rightarrow C_i,$$

where C_i is a predicate in the variables on which v depends. When v can take values in a very large (even infinite) set, the hypotheses $v = a_i$ are replaced by predicates defining a (finite) partition on the range of v; for example, when v has a numeric value, each predicate will define an interval. The appropriate intervals can often be computed automatically from values with which v is compared in the specification.

Currently, only the part of the algorithm for generating invariants from a mode transition table is implemented in SCR*. The full algorithm, which ultimately will be implemented in SCR*, includes methods for generating invariants from event tables and condition tables and a strengthened method for mode transition tables, and can be executed by hand.

The initial application of the automatic invariant generator to the mode transition table for the CD mode class `smOperation` yielded invariants that were unexpectedly weak. This led us to examine the mode transition table more closely and to correct the formulation of several of the events leading to a transition. A typical invariant generated automatically from the revised table is shown in Figure 6. After applying the automatic invariant generator, other parts of the full invariant generation algorithm, including the strengthened method for mode transition tables and the method for event tables, were applied by hand to tables in the

No.	Informal	Formal
7	In Configuration mode, the background is healthy and backup power is not overvoltage	smOperation = sConfiguration ⇒ mHealthyBackground AND mBackupPower ≠ overvoltage

Figure 6. A typical generated invariant for the mode class `smOperation`.

specification.

Although the automatically generated invariant properties of a specification are not the strongest possible invariants, they are often sufficient to establish interesting safety properties [16]. Applying the invariant generation algorithm to CD did not provide results strong enough by themselves to establish the security properties we wished to verify. However, the generated invariants played an extremely useful role: they provided every auxiliary lemma we needed to complete the proofs of all valid security properties that we investigated. Although there is no guarantee that this will always happen, that it did happen for CD suggests that applying automatic invariant generation is an important first step in verifying a set of properties, particularly since, once fully implemented in SCR*, generating invariants will be a push-button technique.

In the case of CD, a total of five auxiliary invariants were required in the proofs of the three (of seven) valid properties of interest we studied that could not be proved automatically using TAME (see Section 3.6) or the SCR* validity checker (see Section 3.7). Of these five invariants, two of which are shown in Figure 7, the first three, including invariant 1 in Figure 7,

No.	Informal	Formal
1	In Configuration mode, backup power is not overvoltage	smOperation = sConfiguration ⇒ mBackupPower ≠ overvoltage
5	If CD is in Off mode, then key 1 in keybank 1 is 0	smOperation = sOff ⇒ cKeyBank1Key1 = 0

Figure 7. Example auxiliary invariants needed in CD proofs.

are subsumed by invariants generated by the implemented algorithm. For example, invariant 1 is subsumed by invariant 7 in Figure 6, which follows from the fragment of the mode transition table of CD in Figure 8 describing the transitions into and out of the mode `sConfiguration`. A fourth auxiliary invariant was proved in TAME from the first three invariants; however, this fourth invariant also follows immediately from additional invariants proved by the strengthened algorithm for mode transition tables (which takes the first three invariants into account). A fifth invariant, invariant 5 in Figure 7, follows from applying the algorithm to the event table of the integer-valued variable cKeyBank1Key1 (Figure 1), which generates the invariant in the equivalent form:

$$\texttt{cKeyBank1Key1} \neq 0 \Rightarrow \texttt{smOperation} \neq \texttt{sOff}$$

(note the use of a partition of the numeric domain of cKeyBank1Key1 in the hypothesis).

Figure 8. Mode transition table fragment.

3.5 Model Checking Properties

When a software specification is represented as an automaton, as in SCR, one can model check its properties. Model checking performs an exhaustive search of some representation of the state space of the automaton. When there is a large number of state variables, and particularly when—as is common in software specifications—the individual variables take values in a large (even infinite) set, the state space can become extremely large, making direct exhaustive search of the entire space difficult or impossible. This is referred to as the *state explosion* problem. The problem can be alleviated by *abstraction*.

For SCR*, we have developed automatable abstraction methods that reduce the state space either by eliminating variables (*variable restriction* methods) irrelevant to a particular property, or by reducing the size of their range of values (*variable abstraction* methods) [11]. However, even with the use of abstraction, the state space to be searched often remains too large to search exhaustively. As a result, model checkers are seldom used to verify that a particular property holds for an automaton. Nevertheless, when a property does not hold for the automaton, a partial search of the state space can often find states that violate the property. In addition to finding states in violation, most model checkers produce counterexamples in the form of scenarios—sequences of events—that lead to the bad state. Such counterexample scenarios help users understand the reasons for property violations, and how to fix the specification to eliminate them. Below, we refer to counterexample scenarios simply as *counterexamples*.

For analyzing properties of an automaton, model checking and theorem proving have complementary strengths. In contrast to a model checker, a theo-

rem prover does not generate a counterexample when a property does not hold. Instead, it generally hits one or more dead-ends in its attempt to prove that the property is an invariant. These dead-ends are in the form of a transition of the automaton that does not (a priori) preserve the property. The user of the theorem prover must then determine whether the transition is reachable in the automaton. It is sometimes possible to prove that the transition is unreachable, usually by showing that the first state in the transition pair violates a known invariant. When the property one is trying to prove does not hold, there will be at least one problem transition for which this cannot be done. However, to demonstrate that the property does not hold, one wishes to generate a counterexample. This cannot be done using the theorem prover itself, but requires either user inspection or support from another tool such as a model checker.

Therefore, applying a model checker to check the validity of a property before trying to establish the property with a theorem prover is generally a good screening strategy, especially since model checking is essentially push-button. If the property is false, the model checker may directly produce a counterexample and save the effort of attempting to prove a property that does not hold and then generating a full counterexample from a dead-end in a proof. In checking security properties for CD, we followed this strategy.

Some example security properties that the SCR CD specification satisfies are shown in Figure 9. Before we

No.	Informal	Formal
1	If CD is tampered with, then key 1 in keybank 1 is zeroized	@T(mTamper) $\Rightarrow$ cKeyBank1Key1′ = 0
2	No key can be stored in location 1 of keybank 1 before an algorithm has been loaded into the first location of algorithm storage segment 1	cKeyBank1Key1 $\neq$ 0 $\Rightarrow$ cAlgStoreSegment1 $\neq$ 0
3	If backup power has an undervoltage when primary power is unavailable, the CD enters either Alarm mode or Off mode	@T(mBackupPower = undervoltage) WHEN mPrimaryPower = unavailable $\Rightarrow$ smOperation′ = sAlarm OR smOperation′ = sOff

Figure 9. Sample true properties for SCR CD.

tried to prove these or any CD security property with TAME (see Section 3.6), we first used the Spin model checker to look for violations of the property. For each property, we used the SCR* tool to automatically extract an abstraction of the CD specification based on the property of interest; this method removes all variables not relevant to determining the validity of the property (i.e., variable restriction from [11]). Then, by hand, we further abstracted the specification by limiting the range of values that certain variables could assume (i.e., variable abstraction from [11]). In our CD study, the reduced variable dependency graphs of the abstractions for different properties varied very little. Figure 10 shows a typical reduced dependency graph, in which the total number of variables has been reduced by 28%, from 39 to 28.

We used Spin several times to analyze each property,

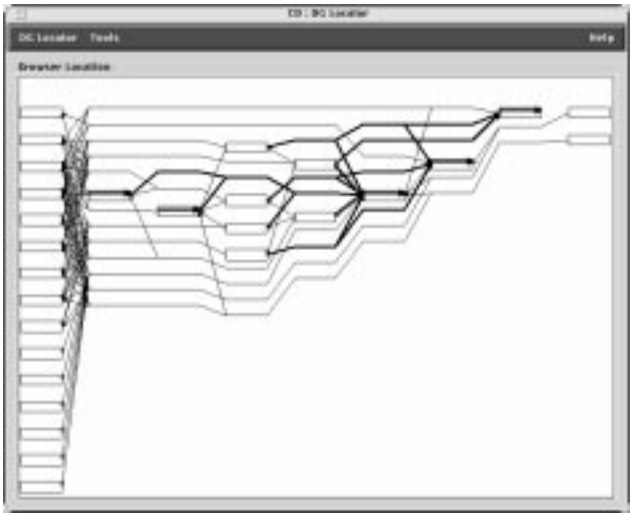

Figure 10. Dependency graph for a typical abstract CD specification for model checking.

adjusting Spin's parameters in an attempt to explore the largest possible subset of the reachable state space. The discovery of property violations led to a few corrections in the formulation of some of the properties. We were unable to find any violations of the properties subsequently proved by TAME. However, since we were not able to search the complete state space of any of the abstract specifications, theorem proving was required to establish the property as an invariant. The importance of the theorem proving phase was demonstrated when we were able to establish with the help of a theorem prover that one property for which Spin was unable to find a violation is not an invariant (see Sections 3.6 and 3.7).

3.6 Checking Properties with TAME

The tool TAME provides an interface to PVS for proving properties of automata models. TAME's goal is to reduce the human effort required in using PVS to specify these automata models and to prove state invariant properties for the models. TAME was originally designed to specify and reason about Lynch-Vaandrager (LV) timed automata [20] but has been adapted to I/O automata [19] and the automata model underlying SCR [2]. TAME provides approximately twenty specialized strategies that implement proof steps mimicking the high-level proof steps typically used by humans in proving invariant properties. Experience has shown that for automata models whose state variables have simple types (such as numerical, boolean, or enumerated types), nearly all state invariants can be proved using the TAME steps exclusively.

We have begun integrating TAME into SCR* by developing a scheme for translating the tables of an SCR specification into a TAME specification. A prototype automatic SCR-to-TAME translator has been implemented [2]. Based partly on our experience with CD, our translation scheme has been refined to produce

more efficient TAME encodings of SCR specifications, i.e., encodings that increase the efficiency of proofs of properties of an SCR specification from its TAME representation. The improvements in the translation scheme take advantage of information from the SCR* consistency checker. Specifically, disjointness checks are used to determine the most efficient representation in TAME of the tables for the variables, and new-state dependencies among the variables are used to find the most efficient representation of the transition relation.

Experience has shown that for many SCR automata—in particular, those not involving timing constraints or other complexities such as tolerances for controlled quantities—a single TAME strategy can automatically prove many state invariants. As indicated in Section 2.1, most invariant properties of interest for an SCR automaton are either properties of a single state of the automaton (one-state properties) or properties of the pair of states in a transition of the automaton (two-state properties).

Technically, both one-state properties and two-state properties can be formulated as state invariants. For a one-state property, this is obvious. A two-state property is expressed as an invariant of a given state using universal quantification of the property over all possible (enabled automaton action, successor state) pairs for the given state. One-state properties must either be proved by induction or by appealing to other one-state properties. Using induction, a proof is broken down into a base case, in which the property is shown to hold for every initial state, and a case for each automaton action (i.e., input event), in which the property is shown to be preserved by the corresponding state transition. Unlike one-state properties, two-state properties are seldom proved by induction, since the transitions possible from any given state seldom have any connection to the transitions possible from one of its successor states. Rather, two-state properties are normally proved by reasoning directly about the transition relation of the SCR automaton.

In TAME, the strategy SCR_INDUCT_PROOF performs the standard parts of an induction proof for a one-state property, and SCR_DIRECT_PROOF does the same for a two-state property. A universal invariant proof strategy is obtained by probing the form of the invariant to see whether it is a one-state or two-state property, and then applying either SCR_INDUCT_PROOF or SCR_DIRECT_PROOF as appropriate. Like the SCR-to-TAME translation scheme, the strategies SCR_INDUCT_PROOF and SCR_DIRECT_PROOF have been refined to significantly improve their average efficiency. For example, TAME's proof by induction of property 3 in Figure 9 originally took about 7900 seconds; with our improved strategies, that time has been reduced to about 400 seconds. In addition, the (non-induction) TAME proof of property 2 in Figure 9, which originally took about 380 seconds, now requires only 37 seconds. Because of the initial long proof times, the first three properties in Figure 9 took a few days to prove. Once the automatic proof strategies were refined, the last three properties in Figure 9 took less than an hour to prove. The fourth property took longer—about 2 days—because we needed to discover and prove two layers of auxiliary invariants needed in its proof. This time would have been greatly reduced if the full invariant generation algorithm (see Section 3.4) had been automated.

When TAME's universal invariant strategy fails to complete the proof of an invariant, there are two possible reasons: either the invariant is false, or additional invariants are needed in the proof. Associated with every "dead-end" in the proof is a problem transition. For one-state properties, this is the transition of the action case in the induction proof in which the dead-end appears. For two-state properties, this is the transition from the given state via some enabled automaton action to the successor state; the strategy SCR_DIRECT_PROOF produces only dead-ends in which the action is known, and hence for deterministic SCR specifications, the successor state (in terms of the given state) is known. TAME provides several analysis strategies that cause PVS to display the details of any problem transition. Eventually, this PVS display will be automatically translated into SCR notation, so that an SCR* user will not have to master the alternate PVS notation. Once the details of the problem transition are understood, the user can decide whether the transition is legal—in which case, the property is false—or whether it is illegal, either because it would violate some two-state property, or because one or the other of the states in the transition would violate some one-state property.

Applying abstraction to a specification is less critical for theorem proving than for model checking. Since a theorem prover can reason about abstract values, reducing the range of a variable using variable abstraction results in little or no improvement in the number of cases the theorem prover must consider. However, eliminating variables can reduce both the number of cases and the complexity of reasoning about state transitions. Therefore, as we did with model checking, we applied variable restriction to the specification prior to undertaking the proof of any property in TAME. Since the abstractions for the individual properties differed very little, we used the same abstraction for all.

Applying the automatic invariant strategy of TAME to seven proposed properties for CD resulted in the automatic proof of four of these properties. For two of the remaining properties, an auxiliary invariant was proposed, proved automatically, and then applied to complete the proof. For example, property 1 in Figure 9 requires an appeal to invariant 5 in Figure 7 in its proof. For the third remaining proposed property, property 3 in Figure 9, an auxiliary invariant was pro-

posed that completed the proof. However, applying the automatic proof strategy to the auxiliary invariant resulted in several dead-ends, and the proposal of three further auxiliary invariants. These three new invariants were then proved automatically.

All auxiliary invariants required in the proofs of the three CD properties which TAME did not prove automatically were generated either by the automatic invariant generator or by simple general extensions to the invariant generation algorithm. Thus, once the invariant generator is extended and communication between the invariant generator and TAME through updates to the SCR specification is made possible, it will be possible to further extend the class of invariant properties that can be proved automatically using TAME. Note that any property that is not a true state invariant will result in one or more dead-ends when TAME's automatic invariant strategy is applied. While TAME supports interactive use to complete the proofs of invariants that do not prove automatically, the substantial effort that can be required for interactive involvement suggests that it is best to use completely automatic tools such as model checkers first to uncover errors in the formulation of properties before applying a general-purpose theorem prover, e.g., through TAME.

TAME also succeeded in finding several (14) problem transitions for a proposed CD property, shown in Figure 11, for which Spin was unable to produce

Informal	Formal
If CD is in Alarm mode, then key 1 in keybank 1 is 0	smOperation = sAlarm ⇒ cKeyBank1Key1 = 0

Figure 11. A property false for SCR CD.

a counterexample. Some intelligent exploration using the SCR* simulator led to the discovery of a counterexample leading to one of these transitions, thus establishing that the property does not hold in our SCR specification of the CD. Examination of the feedback from TAME shows that there are no obvious invariants that forbid the other problem transitions, so that it is likely that they also correspond to counterexamples.

3.7 Applying the Validity Checker

The SCR* validity checker [4] checks the validity of (first-order) one-state or two-state properties directly by using an initial term-rewriting phase followed by application of a decision procedure that combines the use of BDDs (binary decision diagrams) for propositional formulae with a constraint solver for simple integer arithmetic formulae (Presburger formulae). The variable ordering heuristic for the BDDs has been refined to be particularly efficient for SCR specifications. The validity checker can also perform an induction proof of a property by first applying a preprocessor to generate the appropriate base and induction cases and then applying the direct method to the generated cases.

A prototype translator of SCR specifications into input for the validity checker has been built, and the validity checker has been applied to many of the same examples to which TAME has been applied, including the CD properties (after abstraction, as with TAME). The run time required by the validity checker to check the validity of the CD properties was about half the total time taken to run the TAME proofs. For the false property in Figure 11, the validity checker produced a problem transition that was a special case of the one found by TAME that was shown to correspond to a real counterexample, and was in fact the problem transition for which the SCR* simulator was used to discover a corresponding counterexample. Thus, in the same manner as TAME, the validity checker helped to demonstrate that the property is invalid. Unlike TAME, the validity checker cannot be used interactively. Therefore, the CD properties whose proofs required auxiliary invariants were checked after first including all necessary auxiliary invariants as assumptions, rather than by interactively invoking an analog of TAME's APPLY_INV_LEMMA strategy.

Thus, the validity checker can provide an efficient first screening for invariance for any property of an automaton that involves only propositional logic, simple integer constraints, and universal quantification over the states. An extension to handle simple constraints over the rational numbers is planned. To mechanically check the validity of properties involving non-linear numerical constraints, numerical constraints over real numbers, properties whose proofs require types of higher-order reasoning other than induction over reachable states, or properties otherwise requiring interactive, user-guided proofs, access to a general-purpose theorem prover (such as PVS through TAME) is required. The current SCR* consistency checker [13] successfully analyzes approximately 95% of the disjointness and coverage checks in specifications, but, unlike the validity checker, has limited ability to evaluate linear integer constraints. A future version of the validity checker will eventually be used to perform these checks and is expected to improve on this percentage.

3.8 Generating Test Sets for CD

Applying the formal techniques described above will produce very high-quality requirements specifications. Although such high-quality requirements specifications are valuable, the ultimate objective of the software development process is to produce high-quality *software*, software that satisfies its requirements. To weed out software errors and to convince customers that the software performance is acceptable, the software needs to be tested. An enormous problem, however, is that software testing, especially of safety-critical systems, is extremely costly and time-consuming. It has been estimated that current testing methods consume between 40% and 70% of the software development effort [3].

One benefit of the SCR method is that the high-quality specification produced by the method can play a valuable role in software testing. Recently, we developed an automated technique [8] that constructs a suite of test sets from an SCR requirements specification. (Each *test set* is a sequence of system inputs in which each input is coupled with the required system response, i.e., the required system outputs.) To ensure that the test sets "cover" the set of all possible system behaviors, our technique organizes all possible system executions (i.e., traces) into equivalence classes and builds one or more test sets for each class. These test sets can then be used to automatically evaluate the implementation software. By eliminating the human effort needed to build and to run the test sets, such an approach can reduce both the enormous cost and the significant time and human effort associated with current testing methods.

Our technique uses a model checker to construct the suite of test sets from the requirements specification. It does so by using the requirements specification both to generate a valid sequence of inputs and as an *oracle* that determines the required system response, i.e., the set of outputs the system is required to generate for each input event. To obtain a valid sequence of inputs, the input sequence is constrained to satisfy the conditions in the environmental assumptions dictionary of the SCR requirements specification.

We have built a prototype tool in Java that automatically translates an SCR specification into the language of either of two model checkers, executes the model checker to build the test sets, analyzes its outputs, and finally produces a file containing the generated test sets. Our prototype tool has been applied to a number of specifications, including a moderate size specification (containing 55 variables) of a component, WCP_1, of a contractor-specified weapons system [11]. The time needed to construct the test sets was short, varying from a minute or two for small specifications to eight minutes for the WCP_1 specification. The tool generated between three and ten test sets per specification. For the moderate-sized WCP_1 specification, the tool generated 10 test sets, each containing from 2 to 1309 input events. Given the tool's early success in constructing test sets efficiently, we expect that applying the tool to the CD specification should be equally successful. The CD project manager has expressed significant interest in using test sets generated by our tool to test the CD software and other related software. Hence, the next crucial step in our study is to apply the testing tool to the CD requirements specification.

4 Discussion

4.1 Applying Many Formal Techniques

This study illustrates the value and relatively low cost of providing assurance that a COMSEC device is secure by developing a formal specification of its behavior and applying a wide range of analysis techniques to that specification. Each individual technique can provide feedback useful in developing a correct operational specification, formalizing its desired properties, or checking whether the specification satisfies the properties. In the CD study, the process of formal specification and the associated consistency checks revealed our need for an additional mode (`sOff`) and some instances of missing cases in the specification. The fact that the invariants initially produced by the invariant generator were not as strong as expected revealed that several of the events in our initial version of the mode transition table were formulated incorrectly. Model checking was also useful in improving the formulation of properties. Finally, theorem proving not only established the validity of many properties but was helpful in discovering a counterexample for one property. The reason that this property is not true for our SCR representation of the CD is that the prose document does not state explicitly all the implications of being in Alarm mode. This illustrates the benefit of translating a prose specification into a formal language such as SCR, and confirms an observation of Easterbrook and Callahan [6]: inadvertent omissions and the vagueness inherent in prose specifications are exposed both by the process of translating prose into a formal specification and by the application of analysis techniques to the formal specification.

There are several additional benefits of using many techniques. First, one can select the cheapest, most efficient technique for each type of analysis. Second, results established by techniques that are less costly in time and human effort can be used to make the application of more costly techniques more efficient. For example, results from the consistency checking phase can be used to make the encoding of the specification for a theorem prover more efficient, errors found in simulation can lead to corrections in the specification prior to developing formal proofs, counterexamples to properties found by model checking can prevent time wasted in trying to prove invalid properties interactively, and automatically generated invariants can be useful lemmas in establishing properties using a theorem prover. Finally, though individual analysis techniques have particular strengths and weaknesses, with many techniques available, one can use the strengths of one to compensate for the weaknesses of another. For example, a model checker has the ability to provide a counterexample when it discovers that a property is invalid, something a theorem prover cannot do. However, for software specifications, the state space is frequently too large to examine exhaustively to assure that there are no counterexamples, and thus theorem proving techniques are needed for establishing that a property is valid.

The benefit derived from applying the entire suite of analysis techniques in SCR* to a specification can

be increased by extending the automated support for several techniques and by providing automated support for the communication of information between the tools. The full invariant generation algorithm requires implementation. Both the SCR* validity checker and TAME require automated support for building counterexamples when a proof fails. The ability to communicate invariant properties established by one analysis technique to the tools supporting other techniques would, for example, permit TAME to make direct use of lemmas created by the invariant generator.

4.2 Handling Complex Specifications

As noted in Section 3.1, our SCR specification of CD has 39 variables, and the relationships among these variables is complex. In any state after the initial state, the monitored variable mHostCommand can take one of 17 values, and therefore, in any state of the CD, there are 16 possible input events involving changes in this variable. In addition, there are 17 other input variables. As a result, the mode transition table is large, involving 55 events to define 25 mode transitions, and many event tables in the specification are also large: the average number of events per table is 8, with the largest table containing 16 events.

In spite of this complexity, the total time taken in this study to develop and analyze the SCR CD specification was approximately one person-month.[1] As noted above, formalizing the specification of CD in SCR, including using feedback from automatic invariant generation to correct the specification, took only two person-weeks, and even the most complex automated consistency checks ran in minutes. The graphical front-end for simulation of CD was constructed in one day. Improvements to formulation of the properties based on feedback from the model checker took only a few days. The analysis techniques for properties underwent significant improvement during this case study, and as a result, analyzing a property with these techniques now takes at most a few minutes, and sometimes only a few seconds. The most expensive part of the analysis of a property is analyzing deadends to discover needed auxiliary invariants or counterexamples. SCR*'s tool for automatic invariant generation has the potential to alleviate the problem of discovering helpful auxiliary invariants.

5 Related Work

Other mechanized formal methods in which a system is specified as an automaton include RSML [9] and the NRL Protocol Analyzer [21]. RSML has many similarities to SCR: RSML specifications include a set of tables, and there is automated support for checking consistency and a version of completeness (called d-completeness). There are also important differences. RSML explicitly supports specification features, such as hierarchical states and local variables, that are not explicitly supported in SCR (though similar effects can be accomplished using SCR). The AND/OR tables in RSML specify details of *transitions*, while SCR tables specify how *dependent state variables* are updated. Because an automaton has many more transitions than state variables, this means that an RSML specification of a system contains many more tables than an SCR specification of the same system. Automated support for the analysis of specification properties beyond consistency and completeness is not yet extensive. However, RSML has been successfully applied to finding errors in the specification of a complex real life avionics system: the Traffic alert and Collision Avoidance System II (TCAS II).

The NRL Protocol Analyzer is a tool for proving security properties of cryptographic protocols. The environment in which a protocol operates, including the participants in the protocol, is represented as an automaton. The Analyzer performs an exhaustive search of some superset of the reachable states of the automaton, looking for all paths to some given insecure state. The user of the Analyzer first proves lemmas that help to restrict the superset of the reachable states to a set amenable to exhaustive search. The extent to which SCR* can be usefully applied to the same problem domain is a topic for future study.

Other work whose goal is to provide high assurance for security devices includes [25, 17]. Reference [25] describes the methodology used in developing another COMSEC device: the External COMSEC Adaptor (ECA). The development, from modeling the device through implementing and verifying its design, was done using the high-level SCR method. The operational requirements were specified using SCR tables, and the critical requirements model—the desired properties—was specified using the Communicating Sequential Processes language CSP. The development involved both formal and informal transitions between stages, with some automated support from another mechanized theorem prover. Reference [17] describes the design and correctness assurance argument for a device to prevent unauthorized use of a workstation. The assurance argument is made entirely on paper. However, the design should be amenable to specification and verification using the SCR* toolset.

6 Conclusions and Future Work

An important question which this case study investigated is whether it is feasible, low-cost, and of practical value to apply formal methods to establish important properties of operational requirements specifications of communications security devices such as CD. Our results have been encouraging. By using the mechanized formal methods and techniques in the toolset SCR*, we were able to formalize the CD specification and

[1] Some additional time was taken to make improvements in some of the SCR* analysis techniques inspired by the CD example.

establish specification properties, including both completeness and consistency (by applying coverage and disjointness checks) and several desired security properties, with approximately one person-month of effort. We plan to continue this work by specifying additional features of CD such as the cryptographic state, and re-applying SCR* to the extended specification and to new properties involving the additional features.

Various improvements and enhancements to the SCR* toolset are planned. For increased efficiency and portability, the toolset is being ported to Java. Planned improvements include increasing the interoperability of the tools by automating communication between them via updates managed by the specification editor, and improvements in the user feedback concerning possible counterexamples from the SCR* validity checker and from TAME. Planned extensions include implementation of the full invariant generation algorithm and addition of a tool to generate Java code from specifications.

Acknowledgements

We wish to thank Stanley Chincheck and Thomas Sasala for providing us with their prose specification for CD, and both Stan and Tom and our colleague Bruce Labaw for many helpful discussions. Our colleague Ralph Jeffords executed by hand those parts of the invariant generation algorithm that are not yet mechanized. Our colleagues Ramesh Bharadwaj and Steven Sims applied the SCR* validity checker to the CD properties, and Ramesh Bharadwaj discovered a counterexample corresponding to the problem transition found by this tool for the false property described above, thus also providing a counterexample for one of the problem transitions found by TAME. Stuart Faulk, Ralph Jeffords, and Ramesh Bharadwaj gave us helpful comments on early versions of this paper.

References

[1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, pages 192–203. IEEE Computer Society Press, 1996.

[2] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers 1998 (UITP '98)*, Eindhoven, Netherlands, July 1998.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

[4] R. Bharadwaj and S. Sims. Salsa: Combining decision procedures for fully automatic verification. Draft.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 1997.

[7] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, Sept. 1992.

[8] A. Gargantini and C. Heitmeyer. Automatic generation of tests from requirements specifications. In *Proc. ACM 7th Eur. Software Eng. Conf. and 7th ACM SIGSOFT Symp. on the Foundations of Software Eng. (ESEC/FSE99)*, Toulouse, FR, Sept. 1999.

[9] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[10] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[11] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, Nov. 1998.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. 1999. Draft.

[13] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[14] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[16] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, November 1998.

[17] C. E. Landwehr. Protecting unattended computers without software. In *Proc. Thirteenth Ann. Computer Security Applications Conf.*, pages 274–283, San Diego, CA, December 1997.

[18] R. R. Lutz and H.-Y. Shaw. Applying the SCR* requirements toolset to DS-1 fault protection. Technical Report JPL-D15198, Jet Propulsion Laboratory, Pasadena, CA, Dec. 1997.

[19] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[20] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[21] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[22] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[23] D. L. Parnas, G. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.

[24] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.

[25] C. N. Payne, A. P. Moore, and D. M. Mihelcic. An experience modeling critical requirements. In *Proc. COMPASS '94*, pages 245–256, Gaithersburg, MD, June 1994. IEEE Press.

[26] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.